The Roots of LISP

A paper by Paul Graham
reconstructing McCarty's original LISP interpreter
presented by Marco Valtorta

November 28, 2001

An (S-)expression is either an atom or a list.

atom: a sequence of letters

list: is a sequence of (S-)expressions in parentheses and separated by space

atom ex: foo

list examples (foo bar)
(foo (bar))

Seven Primitive Operations

If an S-expression is a list, the first element of the S-expression is the operator, and the other elements are the arguments.

1. quote

```
> (quote (foo bar))
(foo bar)
> (quote a)
a
> 'a
a
```

2. atom

```
> (atom '(foo bar))
()
> (atom '())
t
> (atom 'a)
t
```

3. eq

```
> (eq 'a 'a)
t
> (eq 'a 'b)
()
> (eq '() '())
t
```

4. car (head of)

```
> (car '(foo bar))
foo
```

5. cdr (tail)

```
> (cdr '(foo bar))
(bar)
> (car (cons 'a '(b c)))
a
```

6. cons (list constructor)

```
> (cons 'a '(b c))
(a b c)
```

7. cond

L-evaluated

(cond $(p_1\ e_1) \cdots (p_n\ e_n)$)

```
> (cond ((eq 'a 'b) 'first)
        ((atom 'a) 'second))
second
```

p_1 p_2 e_2

2

```
> ((lambda (x) (cons x '(b))) 'a)
```

2 parameter 3 argument is evaluated 1

[This is the same as (cons 'a 'b)]

```
(a b)

> ((lambda (x y) (cons x (cdr y)))
   'z
   '(a b c))
```

[(cons 'z (cdr '(a b c)))]

[(cons 'z (b c))]

```
(z b c)

> ((lambda (f) (f '(b c))
   '(lambda (x) (cons 'a x)))
```

[((lambda (x) (cons 'a x)) '(b c))]

```
(a b c)
```

A Notation for Functions

- `lambda` notation "A form can be converted into a function if we can determine the correspondence between the variables occurring in the form and the . . . arguments of the desired functions" [McCarthy, p.186].

```
((lambda (p1 ... pn) e) a1 ... an)
```

- `label` notation "The lambda notation is inadequate for naming functions defined recursively," because there is no way to name a function within itself.

```
(label f (lambda (p1 ... pn) e))
```

```
(defun f (p1 ... pn) e)
```

```
>(subst 'm 'b '(a b (a b c) d))

(a m (a m c) d)

(label subst (lambda (x y z)
               (cond ((atom z)
                      (cond (eq z y) x)
                            ('t z)))
                     ('t (cons (subst x y (car z))
                               (subst x y (cdr z))))))))

(defun subst (x y z)
  (cond ((atom z)
         (cond (eq z y) x)
               ('t z)))
        ('t (cons (subst x y (car z))
                  (subst x y (cdr z))....)
```

Some Functions

((lambda x (eq x '())) '(foo bar))

- null.

(lambda x (eq x '())) To apply:

```
(defun null. (x) (eq x '()))
```

- and.

```
(defun and. (x y)
  (cond (x (cond (y 't) ('t '())))
        ('t '())))
```

- not.

```
(defun not. (x)
  (cond (x '())
('t 't)))
```

if x then false else true

("negation (by) as failure")

Some Functions, ctd.

- append.

```
(defun append. (x y)
  (cond ((null. x) y)
('t (cons (car x) (append. (cdr x) y)))))
```

- pair. (pairup. or zip)

```
(defun pair. (x y)
  (cond ((and. (null. x) (null. y)) '())
((and. (not. (atom x)) (not. (atom y)))
 (cons (list (car x) (car y))
       (pair. (cdr x) (cdr y))))))
```

Some Functions, ctd.

assoc.

is a list of pairs as built by pair.

```
(defun assoc. (x y)
  (cond ((eq (caar y) x) (cadar y))
        ('t (assoc. x (cdr y)))))
```

Think of y as a "dictionary" or "environment"

(caar y) ≡ (car (car y))

(cadar y) ≡ (car (cdr (car y)))

'((w new) (u a) (z b))

our "dictionary"

(caar) = w

(cadar) = new (check!)

Eval

expression

association list (dictionary/environment)

```
(defun eval. (e a)
      (cond
1.    ((atom e) (assoc. e a))
2.    ((atom (car e))
       (cond
        ((eq (car e) 'quote) (cadr e))
        ((eq (car e) 'atom)  (atom (eval. (cadr e) a)))
        ((eq (car e) 'eq) (eq (eval. (cadr e) a)
                              (eval. (caddr e) a)))
        ((eq (car e) 'car) (car (eval. (cadr e) a)))
        ((eq (car e) 'cdr) (cdr (eval. (cadr e) a)))
        ((eq (car e) 'cons) (cons (eval. (cadr e) a)
                                  (eval. (caddr e) a)))
        ((eq (car e) 'cond) (evcon. (cdr e) a))
        ('t (eval. (cons (assoc. (car e) a)
                         (cdr e))
                   a))))
3.    ((eq (caar e) 'label)
       (eval. (cons (caddar e) (cdr e))
              (cons (list (cadar e) (car e)) a)))
4.    ((eq (caar e) 'lambda)
       (eval. (caddar e)
              (append. (pair. (cadar e)
      (evlis. (cdr e) a))
                     a))))))
```

```
        (quote a)
(eval. 'a    '((b d) (a foo)))

e = (quote a)
(car e) = quote
(cadr e) = (car (cdr e)) = (car (cdr (quote a))) =
   = (car (a)) = a
                                   environment
(eval. '(eq 'a 'a)   '())

(eq   (eval. (cadr (eq 'a 'a)     a)
      (eval. (caddr (eq 'a 'a)    a)) = 

= (eq  (eval. 'a     a)      ('a)
       (eval  'a     a))   =

= (eq  (eval (quote a)  a)
       (eval (quote a)  a)) =

= (eq  a
       a)   = t          (Simple b/c we did
                           not need the
                           environment.)
```

(car c) = (c_1 e_1)

(caar c) = (car (car c)) = (car (c_1 e)) = c_1

(cdar c) = (cdr (car c)) = (cdr (c_1 e_1)) = (e_1)

Two Auxiliary Functions

(cadar c) = (car (cdr (car c))) = (car (e_1)) = e_1

evcon. scans the list of (c_i, e_i) pairs, until it finds a condition (say, c_j) that is true in the environment. It then returns the matching expression (e_j), evaluated in the same environment.

c = ((c_1 e_1) (c_2 e_2) ... (c_n e_n))

(caar c) = c_1

(cadar c) = e_1

(eval. c_1 a)

```
(defun evcon. (c a)
 (cond ((eval. (caar c) a)
        (eval. (cadar c) a))
       ('t (evcon. (cdr c) a))))
```

((c_2 e_2) (c_3 e_3) .... (c_n e_n))

evlis. evaluates the list of expression m in environment a.

```
(defun evlis. (m a)
  (cond ((null. m) '())
        ('t (cons (eval. (car m) a)
                  (evlis. (cdr m) a)))))
```

Evaluation of function calls

Calls to functions are evaluated by replacing the atom which is the function name with its value, which is a `lambda` or `label` expression and evaluating the resulting expression. E.g.:

```
cg-user(64): (eval. '(f '(b c))
    '((f (lambda (x) (cons 'a x)))))
```

a (the environment)

is evaluated as:

```
(eval. '((lambda (x) (cons 'a x)) '(b c))
       '((f (lambda (x) (cons 'a x)))))
```

3 2 1

which returns

```
(a b c)
```

Evaluation of label Expressions

"A `label` expression is evaluated by pushing a list of the function name and the function itself on the environment, and then calling `eval.` on an expression with the inner `lambda` expression substituted for the `label` expression" [Graham, p.9]. E.g.:

```
(eval. '((label firstatom (lambda (x)
                     (cond ((atom x) x)
                           ('t (firstatom (car x))))))
         y)
        '((y ((a b) (c d)))))
```

is evaluated as:

```
(eval. '((lambda (x)
            (cond ((atom x) x)
                  ('t (firstatom (car x)))))
         y)
        '((firstatom
           (label firstatom (lambda (x)
                      (cond ((atom x) x)
                            ('t (firstatom (car x))))))
          (y ((a b) (c d)))))
```

x

a new pair in the environment

which returns a.

Evaluation of lambda Expressions

"An expression of the form `((lambda (p1 ... pn) e) a1 ... an)` is evaluated by first calling `evlis.` to get a list of values `v1, ..., vn` of the argument `a1, ..., an` and then evaluating `e` with `(p1 v1), (pn vn)` appended to the front of the environment" [Graham, p.10]. E.g.:

```
cg-user(67): (eval. '((lambda (x y) (cons x (cdr y)))
                      'a
                      '(b c d))
                    '())
```

= initial environment is empty

is evaluated as

```
(eval. '(cons x (cdr y))
       '((x a) (y (b c d))))
```

the environment has the parameter values

which returns

```
(a c d)
```

Examples

The function f is bound to a function that conses `a` to some list. Note that the second argument to eval is the environment.

```
cg-user(64): (eval. '(f '(b c))
                    '((f (lambda (x) (cons 'a x)))))

(a b c)
```

The function `firstatom` finds the first atom of its argument. Argument y is bound to the list `((a b) (c d))`

```
cg-user(66): (eval. '((label firstatom (lambda (x)
                                          (cond ((atom x) x)
                                                ('t (firstatom (car x)))))
                      y)
                    '((y ((a b) (c d)))))
a
```

We evaluate a nameless function of two arguments. The environment is empty. The two arguments are quoted.

```
cg-user(67): (eval. '((lambda (x y) (cons x (cdr y)))
                      'a
                      '(b c d))
                    '())
(a c d)
```

Comments

McCarty's 1960 language does not have:

1. side effects (viz. destructive assignment)

2. sequential execution (which is only useful when one has side effects!)

3. practical numbers

The language has dynamic scope: `label` changes the environment without any concern for the static layout of the program.

McCarthy's original paper describes an implementation of the LISP programming system for the IBM 704. In February 1960, this included some debugging facilities, and a "program feature" supporting destructive assignment, sequential execution and jumps, was available.

(PROG ...)